Software & Tools

TEXML: Resurrecting TEX in the XML world

Oleg Parashchenko

1 Foreword

TEXML is an XML syntax for TEX, LATEX and ConTEXt. This definition is extremely correct, but I dislike its formality. Instead, I prefer the following.

Thanks to TEXML, you can reuse your TEX skills in the XML world. With TEXML, XML publishing becomes a case of TEX publishing.

TEXML is a very simple thing. You can learn it in a minute by looking at the examples in the section 'TEXML tour'. But knowing the syntax isn't enough.

To feel TEXML, you need to know its past and future, the ideas behind it, and understand the author's intentions. That's why the technical stuff is wrapped by the sections with my very subjective view on the topic of XML publishing.

In the most cases, the words 'TEX' and 'LATEX' are interchangeable, and they mean also any other TEX format.

The author is from the XML world. The TEXML home page is `http://getfo.org/texml/`.

2 Why XML, not TEX, why TEX, not XML

The best thing about XML is that everyone knows what it is. XML is ubiquitious now, and especially in the area of technical documentation. Indeed, its parent, SGML, was created to support authoring of technical manuals.

TEX users have different opinions on XML. But nobody rejects the idea of logical markup is very obvious and essential. From the high level point of view, all the markup methods are the same.

What in XML looks like

```
<environment> ...text... </environment>
```

in LATEX looks like this:

```
\begin{environment} ...text... \end{environment}
```

The only difference is notation. But it's a very important difference. Computers prefer XML, humans prefer LATEX.

Among benefits of logical markup is the possibility of single source publishing, when the same source document can be converted to different output formats. XML is the best choice because XML libraries exist in any practical programming language. On the other hand, the only correct TEX parser is TEX itself, and TEX is locked in its sandbox.

On the other side, the ideal XML world isn't ideal. How to get PDF from XML? Theory says that you would write an XSLT (W3C, 1999) program which converts XML to XSL-FO (W3C, 2001), and use an XSL-FO formatter which generates PDF from XSL-FO.

XML+XSLT → XSL-FO → PDF. There are two issues: first, tools, which is hopefully temporary; and second, too much automation, which is fatal.

Only a few tools implement XSL-FO in full, and all these tools are commercial, without open source alternatives (the best one is FOP, which is under development), and the W3 Consortium has started work on XSL-FO 2.0.

But the worst is that the joke *'automatically' means you can't fix it if something goes wrong* applies perfectly to the XSL-FO way. When you need to tune a generated layout, you'll find that XSL-FO level is too low, and editing XSL-FO isn't much better than editing PDF. Also you'll find that XML and XSLT levels are too high and editing here smells bad.

The broken layout isn't a showstopper in LATEX. Your writings are marked up logically, and when you need typographical tunings, you just use low-level primitives.

Time for a short summary:

- XML is good as a markup language,
- TEX is good for publishing documents.

Why not take the best from both worlds? That is, have sources in XML and publish the documents through TEX. But how?

3 XML to TEX — how

When converting XML, there is no better alternative than XSLT. This language is specially designed to convert XML, is based on experiences with the Lisp-like DSSSL language, has a large user and expert base, and has decent support by many tools on many platforms.

Why not Java, or Perl, or Python, or something else? Because XML is alien to them. It's inconvenient to use the traditional languages for processing XML, for either parsing or converting.

For example, in one project the author worked on a Java application. One procedure was more than 20 lines in size, debugged and enhanced several times, and still couldn't be compared in functionality with a small XPath (a part of XSLT) expression of several characters.

Worse, the whole library was a partial, poorly documented, limited re-invention of XSLT. I think it's the doom of any program which converts XML. Instead of using a poor imitation, it's better to use XSLT itself.

The knowledgeable reader can say that XSLT is a language to convert from XML to XML, not from XML to TeX, and ask if XSLT is still so great to generate TeX.

No, I have to answer, converting XML directly to TeX is nightmare. XSLT is very weak and unbelievably verbose in working with strings, but that's what is required when generating TeX code.

What is expected from a TeX code generator:

- escaping special TeX characters (for example '`<`' to '`\<`' or, better, to '`\textless{}`');
- disjoining ligatures ('`---`' isn't the long dash in XML, the long dash is the symbol '`—`');
- mapping from Unicode characters to LaTeX sequences;
- avoiding empty lines, which start a new paragraph in TeX.

And there are common errors when generating TeX code. (See bug databases for such projects as db2latex (Casellas and Devenish, 2004), dblatex (Guillon, 2006) and others.)

- Opening or closing brace is forgotten.
  `<i>some</i> text`
  → `{\it some text`
  instead of `{\it some} text`.[1]
- No space after the command name.
  `{\itsome} text`
- Space instead of braces.
  `here is<i> some</i> text`
  → `here is{\it some} text`
  instead of `here is{\it{} some} text`

If you write a TeX code generator, you should pay attention to everything. You need accuracy and patience, and the work isn't trivial. Therefore you'd prefer to delegate TeXification from your program to something else.

TeXML is the best and probably the only candidate. You create XML, which is much easier, and then a TeXML processor converts TeXML to TeX.

Short summary:

- XSLT is the best tool for converting XML to XML,
- it's better to delegate TeX code generation.

That's why we have TeXML, an XML syntax for TeX/LaTeX/ConTeXt. Conversion from XML to TeX consists of two steps:

- an XSLT program converts XML to TeXML, and
- a TeXML processor converts TeXML to TeX.

TeXML is an XML language with just a few tags, and converting XML to XML is the specialization of XSLT; therefore you need only basic knowledge of XSLT to convert XML to TeX.

4 TeXML tour

The TeXML markup language is minimalistic. Most of the time, you use only three elements: `cmd`, `env` and `group` (the other elements are `pdf`, `math`, `dmath`, `ctrl`, `spec` and `TeXML`).

To get accustomed to TeXML, it's enough to learn the examples presented in this section. The original paper by Douglas Lovell (Lovell, 1999) is also a good introduction, but it's out of date. For a detailed description of contemporary TeXML, consult the TeXML specification (Parashchenko, 2006b).

Installation and usage instructions are on the TeXML home page: `http://getfo.org/texml/`. A pleasant feature is that it's enough to unpack the distribution package to use TeXML. The installation procedure isn't required, it's for convenience only.

4.1 Simple TeXML file

An example of a simple TeXML document:

```
<TeXML>
<TeXML escape="0">
\documentclass[a4paper]{article}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
</TeXML>
<env name="document">
I'm not afraid of the symbols ^,
$, > and others.
</env>
</TeXML>
```

The result of conversion to TeX is the LaTeX document:

```
\documentclass[a4paper]{article}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\begin{document}
I'm not afraid of the symbols \^{},
\textdollar{}, \textgreater{} and others.
\end{document}
```

This example demonstrates:

- the root element is `TeXML`,
- TeX special symbols are escaped automatically,
- it's possible to disable escaping.

By the way, while preparing the original LaTeX example, I made two errors:

- '`\textgreater`' instead of '`\textgreater{}`' (result — no space after the symbol '>'),
- '`\^`' instead of '`\^{}`' (result — the circumflex over the comma instead of the symbol '`^`').

TeXML saves me from such basic errors.

[1] In production we might use `\textit{...}`, but for illustrative purposes here I use `{\it ...}`.

Disabling escaping is not recommended. Usually it's a misuse of TeXML. But to keep examples simple, I do use it for creating the LaTeX header.

4.2 More TeXML

This document uses more TeXML elements:

```
<TeXML>
 <cmd name="documentclass">
   <opt>a4paper</opt>
   <parm>article</parm>
 </cmd>
 ....
 <env name="document">
   Hello, <group><cmd name="it"/>World</group>!
 </env>
</TeXML>
```

After converting to TeX, the result is:

```
\documentclass[a4paper]{article} ....
\begin{document}
Hello, {\it{}World}!
\end{document}
```

This example demonstrates the three most often used TeXML elements:

- `cmd` creates a LaTeX command,
- `env` creates a LaTeX environment,
- `group` creates a LaTeX group.

The example also demonstrates how to create the LaTeX header using regular TeXML instead of disabling escaping.

4.3 Better layout

This example demonstrates how to tune the layout of a generated LaTeX code. The result can be made indistinguishable from code written by a human.

In the last example, we got the following LaTeX document:

```
\documentclass[a4paper]{article} ....
\begin{document}
Hello, {\it{}World}!
\end{document}
```

A better code layout is:

```
\documentclass[a4paper]{article}
....
\begin{document}
Hello, {\it World}!
\end{document}
```

The source TeXML code uses the attributes `nl2` and `gr` to tune the layout:

```
<TeXML>
  <cmd name="documentclass" nl2="1">
    <opt>a4paper</opt>
    <parm>article</parm>
  </cmd>
  ....
  <env name="document">
    Hello, <group>
      <cmd name="it" gr="0"/>World</group>!
  </env>
</TeXML>
```

4.4 PDF literal strings

Let's start with the following LaTeX code:

```
\documentclass{article}
\usepackage[T2A]{fontenc}
\usepackage[koi8-r]{inputenc}
\usepackage{hyperref}
\begin{document}
\section{Заголовок (Title)}
Текст (Text)
\end{document}
```

The code looks fine, but due to the Russian letters, LaTeX raises the errors:

```
Package hyperref Warning:
    Glyph not defined in PD1 encoding,
(hyperref) removing ‘\CYRZ’ on input line 6.
```

For the document above, the solution is to use

```
\usepackagep[unicode]{hyperref}
```

But this solution is not generic. For example, for CJK text, it fails with some obscure error like:

```
! Incomplete \ifx; all text was ignored ...
```

I prefer the universal solution that uses Unicode strings for the PDF names:

```
\documentclass{article}
\usepackage[T2A]{fontenc}
\usepackage[koi8-r]{inputenc}
\usepackage[unicode]{hyperref}
\begin{document}
\section{\texorpdfstring{Заголовок (Title)
}{\004\027\004\060\004\063\004\076\004\073
\004\076\004\062\004\076\004\072\000\040\0
00\050\000\124\000\151\000\164\000\154}}
Текст (Text)
\end{document}
```

Comparing to the previous example, I use

- the option `unicode` for the package `hyperref`,
- the command `texorpdfstring` to assign the name for the PDF bookmark entry.

The content of `texorpdfstring` is created by the TeXML command `pdf`:

```
<cmd name="section">
  <parm>
    <cmd name="texorpdfstring">
      <parm>Заголовок (Title)</parm>
      <parm><pdf>Заголовок (Title)</pdf></parm>
    </cmd>
  </parm>
</cmd>
```

4.5 Encodings

Consider TeXML with the Russian letters:

```
<TeXML>Текст</TeXML>
```

Default translation to LaTeX produces:

```
\cyrchar\cyrT{}\cyrchar\cyre{}\cyrchar....
```

The result is correct, but those who speak Russian prefer to see the real Russian letters instead of TeX commands.

To achieve this, specify the desired output encoding to the TeXML processor using the command line option `--encoding` (or `-e`). When the output encoding is, for example, `koi8-r`, the result is:

```
Текст
```

4.6 ASCII output

The following LaTeX document contains the phrase 'Hello, World!' written in Chinese:

```
\documentclass{article}
\usepackage[encapsulated]{CJK}
\usepackage{ucs}
\usepackage[utf8x]{inputenc}
\begin{document}
\begin{CJK}{UTF8}{cyberbit}
世界，你好！
\end{CJK}
\end{document}
```

LaTeX successfully compiles this document. But imagine:

- you've got a problem with a CJK or other non-latin document,
- latin documents don't have this problem, so
- you want to ask for help.

To get help, you should provide a minimal example to reproduce the problem. Unfortunately, in many cases, your non-ASCII text will be corrupted.

Luckily, TeX provides ASCII sequences to encode non-ASCII bytes. With the command line flag `--ascii` (or `-a`), the TeXML processor uses ASCII sequences. For example, the above LaTeX document is written as follows:

```
\documentclass{article}
\usepackage[encapsulated]{CJK}
\usepackage{ucs}
\usepackage[utf8x]{inputenc}
\begin{document}
\begin{CJK}{UTF8}{cyberbit}
^^e4^^b8^^96^^e7^^95^^8c^^ef^^bc^^8c
^^e4^^bd^^a0^^e5^^a5^^bd^^ef^^bc^^81
\end{CJK}
\end{document}
```

5 History and other TeXMLs

A long long time ago a company for which I consulted had to switch from XML publishing using FrameMaker+SGML to a pure XML publishing using XSL-FO. At the same time, I joined a documentation team for a large open source project. In both cases, we needed an open source XSL-FO formatter, and we didn't find a viable tool.

I had the courage to write my own good open source XSL-FO processor. The idea was that I could build it on top of LaTeX, and I thought I need only a converter from XSL-FO to TeX.

The language to use for the converter was obvious to me: XSLT. Quite soon, I found that writing valid TeX code is hard and unpleasant work. Instead, I got the bright idea that it's better to use an intermediate XML language, and even half-prototyped it.

At some moment I noticed that I had reinvented the wheel. Much earlier, Douglas Lovell presented (Lovell, 1999) his TeXML at the TUG99 conference. Unfortunately, his TeXMLLatté, a Java implementation of TeXML, was 'retired' and not available for download.

But the specification survived. I found that it was very close to my ideas and decided to countinue with the existing solution. As result, all the old TeXML documents are still valid and can be processed by my tool.

In addition to the original Java TeXML, I found processors written in Ruby (isn't available anymore) and Perl (Houser, 2001). Unfortunately, their status was 'works for the author', but I needed production quality.

That's why I started my own TeXML implementation. The choice of Python was quite arbitrary. At that time I was learning this language, and I prefer learning by doing. Now I think it was a fortunate choice, as Python is a very good compromise between popularity and speed of development and running.

The first version just worked and was without any advanced features. However, it found its users, for whom I'm very thankful. The feedback revealed that the nice layout of the generated TeX code is of much greater importance than I considered. I accepted the challenge, and since version 1.1, TeXML writes human-friendly TeX code.

I presented version 1.1 at a Russian conference (Parashchenko, 2004b), and I thought that TeXML development was finished.

Working on a real publishing project, however, I added more features to TeXML, mostly related to internationalization support. Meanwhile, I also investigated how to deal with TeX and XSLT limitations. This activity resulted in the projects sTeXme (Parashchenko, 2004a) (TeX+Scheme) and XSieve

(Parashchenko, 2006c) (XSLT+Scheme), one of the Google Summer of Code 2005 projects, presented at the XTech 2006 conference.

TeXML popularity grew, and I started to get contributions. One of the TeXML users, Paul Tremblay, used ConTeXt for publishing. He added ConTeXt support to TeXML, reworked bits of TeXML code and wrote extensive documentation (Tremblay, 2005) on how to imitate XSL-FO constructions in ConTeXt. That's a must-read for those who are interesed in the topic.

In June 2006, I collected all the improvements, rewrote documentation, packed the whole as a usual Python package and released version 2.0. No bugs reported till now (March 2007).

6 The TeXML processor: present and future

At the moment, the only TeXML processor implementation is written by me in Python. It uses only few standard modules and therefore is portable and can be used anywhere if Python is installed.

The core of the TeXML processor is a standalone Python library, therefore TeXML functionality is available to any Python application. It might be that TeXML is available to Java programs using JPython and to .NET programs using IronPython, but checking this has low priority on my long-term TODO list.

TeXML follows the three-step approach to software development: make it work, make it correct, make it fast. TeXML is currently on the second level, 'work correct', so now it's time to improve performance. The processor works much faster than XSLT, but it can be made an order of magnitude faster yet.

The approach is to use finite automata. The current code escapes the output stream character by character. The set of loops, flags and nested conditions adds an overhead to the processing time. By comparison, with automata the only flags are the current state, the current character, and the table of state changes. Overhead per character is minimized.

The second main benefit of automata is that it would make explicit all the rules how to generate correct TeX code with nice layout. At the moment, this knowledge is hidden inside the spaghetti code, that is hard to maintain and modify.

And I'd like to improve some things. For example, the TeXML

```
<cmd name="command"/><ctrl ch="\"/>
```

is translated to

```
\command{}\\
```

I'd prefer to automatically avoid dummy groups:

```
\command\\
```

Yet another benefit of using automata is that TeXML could be ported to other languages. The non-trivial TeXML logic, were it written as automata in some well-known format, such as S-expressions or XML, could be automatically translated to a code in any language.

Unfortunately, all these wonderful perspectives are for the far far future. I'm satisfied with the current state of TeXML and prefer to concentrate on other projects.

Creating automata for TeXML could be a good master thesis or even a PhD work. If you know someone who might be interested in this task, don't hesitate to mention TeXML.

7 Nice layouts, `diff` and `patch`

Probably you've noticed how much attention I devote to the nice layout of the generated code. But what's the benefit except aesthetic?

Before answering, I'd like to note that aesthetic appearance is indeed a benefit. You know the saying, ugly things can't fly. I believe in it. And definitely, nobody is interested in working with the intermediate ugly code which appears in many other XML-to-PDF-through-LaTeX projects.

Automatically generated PDFs can't be ideal. From time to time, there are layout faults that you'd like to fix. To tune these places, you need to edit the LaTeX code. When this code is ugly and bad, you might prefer to tolerate the faults instead of fixing them. On the contrary, when the code is human-friendly, you are likely to look into the code and fix the problems.

But the main benefit of human-friendly code is that such code is also `diff`- and `patch`-friendly.

Imagine that you've fixed all the layout faults in the LaTeX code. Unexpectedly, a proofreader has updated the source XML. How to generate a new PDF, both with your and the proofreader's changes? The naive user has two alternatives:

- detect what's changed in XML and repeat the changes in the LaTeX code, or
- re-generate PDF and re-apply layout corrections in the LaTeX code.

Both options are miserable, boring and error-prone. Open source software developers would prefer a better way using `diff` and `patch`.

- Take the initial LaTeX file, take the current version with the layout fixes, and generate a patch-file using `diff`.
- Generate a new PDF from the new XML.

- Apply the patch-file to the new LaTeX file and re-generate the PDF.

In most cases, everything goes smoothly and all the changes, from both you and the proofreader, are applied.

Thanks to the good LaTeX code formatting, as produced by TeXML, this way is indeed possible. Instead of saying 'patch-file', I prefer to say 'beauty memory'. It sounds more appealing and descriptive.

To automate this procedure, I developed Consodoc (Parashchenko, 2006a), an XML to PDF publishing tool on top of TeXML. The user's guide for Consodoc is generated by Consodoc itself. Here is an example of the project file:

```
import Consodoc
env = Consodoc.default_process(
  in_file = 'in/guide.xml',
  in_xslt = 'support/guide.xsl'
)
Depends('tmp/guide.pdf', 'support/guide.cls')
```

The project file defines that the source XML file is `in/guide.xml`, TeXML is generated by the XSLT program `support/guide.xsl`, and implicitly defines that the patch file is `in/guide.patch`. It also specifies, explicitly and implicitly, the dependencies of the files: if a file is changed, than all the dependent files should be re-generated. To build PDF, just say on the command line: `cdoc`.

Consodoc is a very new product, but it is already usable and successfully passed unit and functional testing. I recommend Consodoc for use in the production environment by early adopters.

8 Final words

Publishing XML is still a practical problem, even when the quality of the result isn't very important. Different approaches are suggested, from using the XSL-FO standard to developing a custom solution, but the Right Thing is still to appear.

The TeXML approach is one of the candidates. Instead of inventing something new, it smoothly integrates existing successful technologies and experience. First, it uses TeX as the typesetting engine. Second, it uses XSLT as the conversion language.

Third, with the help of the `diff` and `patch` tools, the beauty memory maintains layout corrections of the PDF documents. I'm not aware of any other XML-to-PDF solution with this feature.

The only TeXML problem is the lack of sample conversion scripts. But I've started work on the TeXML stylesheets for DocBook, a popular XML standard for technical books, therefore this problem will be fixed in the near future.

I expect this union — TeXML, beauty memory and DocBook TeXML stylesheets — will have a big impact on XML publishing, causing restoration of the TeX technologies in the modern XML world. Join the TeXML movement!

References

W3C. "XSL Transformations (XSLT). Version 1.0. W3C Recommendation 16 November 1999". See `http://www.w3.org/TR/1999/REC-xslt-19991116`, 1999.

W3C. "Extensible Stylesheet Language (XSL). Version 1.0. W3C Recommendation 15 October 2001". See `http://www.w3.org/TR/2001/REC-xsl-20011015/`, 2001.

Casellas, Ramon, and J. Devenish. "Welcome to the DB2LaTeX XSL Stylesheets". See `http://db2latex.sourceforge.net/`, 2004.

Guillon, Benoît. "DocBook to LaTeX/ConTeXt Publishing". See `http://dblatex.sourceforge.net/`, 2006.

Houser, Chris. "TeXMLapis". Available from `http://bluweb.com/us/chouser/proj/texmlapis/`, 2001.

Lovell, Douglas. "TeXML: Typesetting XML with TeX". *TUGboat* **20**(3), 176–183, 1999.

Parashchenko, Oleg. "sTeXme". See `http://stexme.sourceforge.net/`, 2004a.

Parashchenko, Oleg. "TeXML: an XML vocabulary for TeX". See `http://getfo.org/texml/thesis.html`, 2004b. Thesis for the First International Conference of Open-Source Developers, Obninsk, Russia.

Parashchenko, Oleg. "Consodoc publishing server: XML to beautiful documents". See `http://consodoc.com/`, 2006a.

Parashchenko, Oleg. "TeXML specification". See `http://getfo.org/texml/spec.html`, 2006b.

Parashchenko, Oleg. "XSieve: extending XSLT with the roots of XSLT". See `http://xmlhack.ru/protva/xtech2006-paper.pdf`, 2006c. XTech 2006: Building Web 2.0, 16-19 May 2006, Amsterdam, The Netherlands.

Tremblay, Paul. "Welcome to context-xml". See `http://getfo.org/context_xml/`, 2005.

◇ Oleg Parashchenko
Saint-Petersburg State University,
7-9, Universitetskaya nab,
Saint-Petersburg, Russia
olpa (at) uucode dot com
http://uucode.com/